

Continuous Delivery and Tooling in Go

Benjamin Muschko

O'REILLY®

About the trainer



bmuschko



bmuschko



bmuschko.com



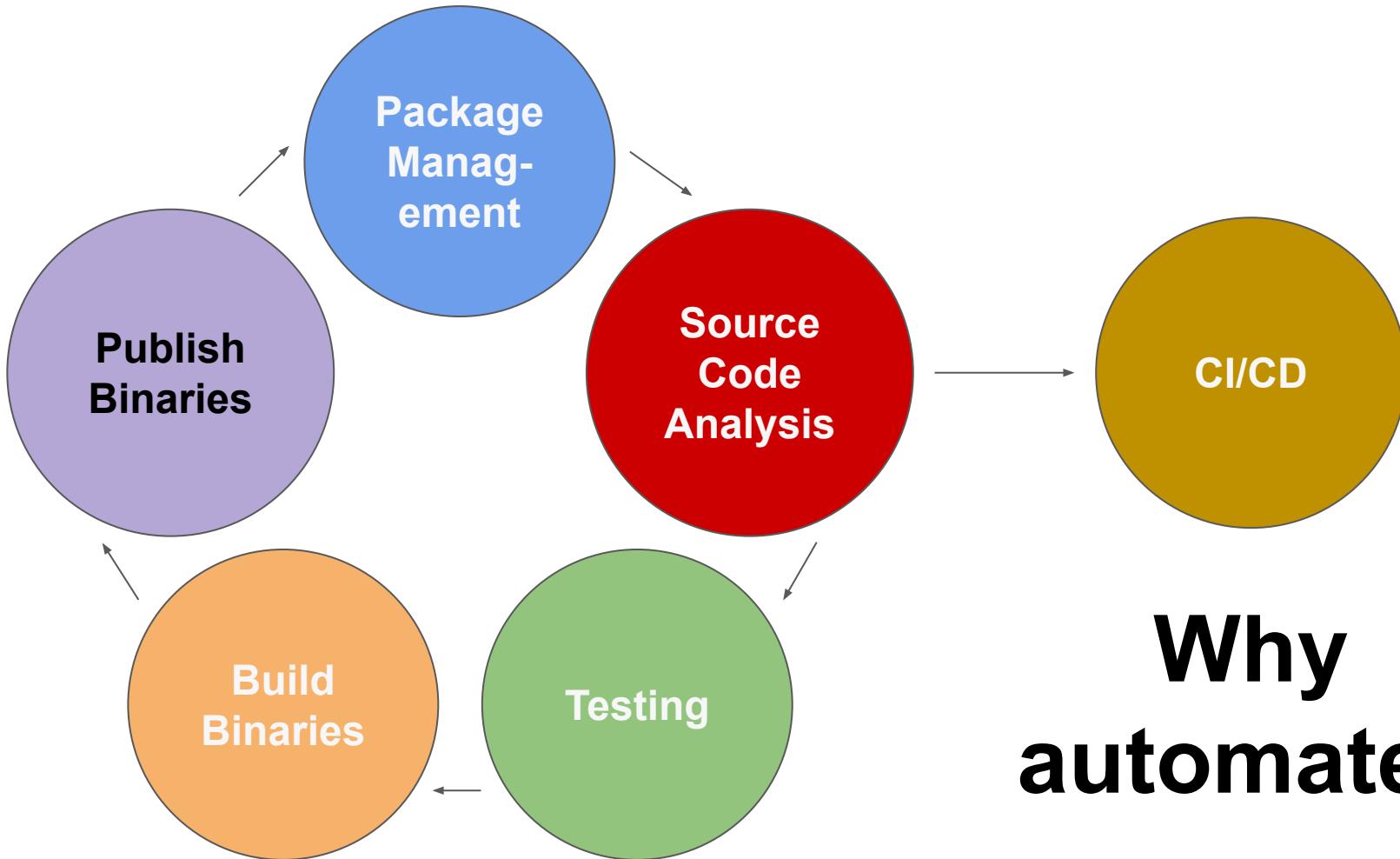


Training Objectives

Agenda

- Source code analysis
- Package management
- Implementing and executing tests
- Building deliverable artifacts
- Publishing deliverable artifacts
- CI/CD for Go projects
- Wrap up and Q&A



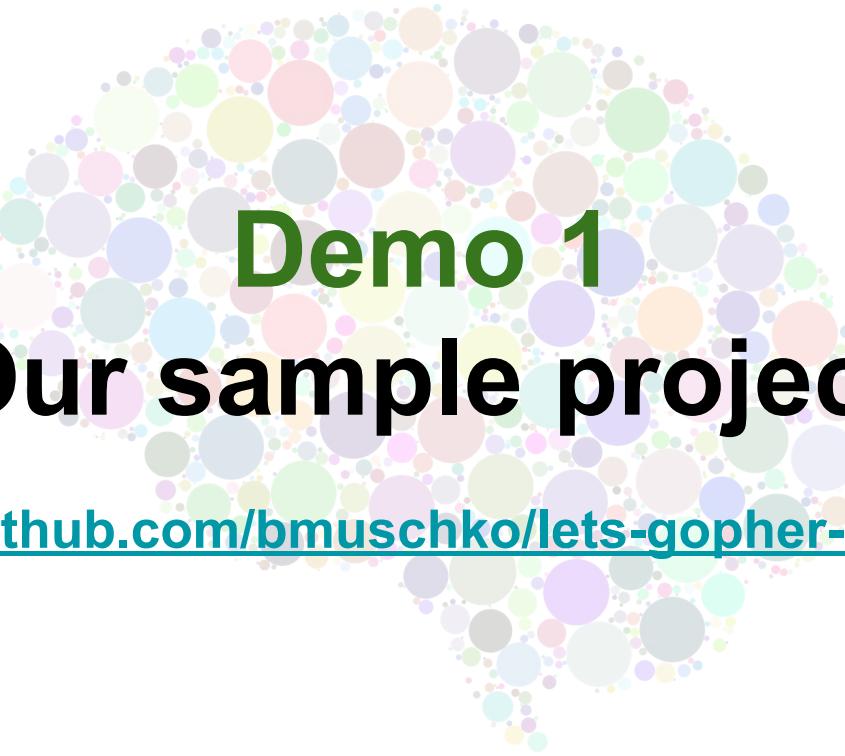


Why automate?



Discussion

**What automation aspects
would you like to see
covered in this training?**



Demo 1

Our sample project

<https://github.com/bmuschko/lets-gopher-exercise>

Source code analysis

1



Discussion

**What analysis tools do
you find useful in your
projects?**

Detecting common mistakes

golang.org/cmd/vet/

Part of the standard, built-in Go tooling

```
$ go vet (mypackage)
```

*Current directory or
provided package*



Demo 2

Using Go vet

Enforcing coding style conventions

github.com/golang/lint

```
$ go get -u golang.org/x/lint/golint
```

```
$ golint
```

*Path to filename,
directory or package*



Demo 3

Using Golint

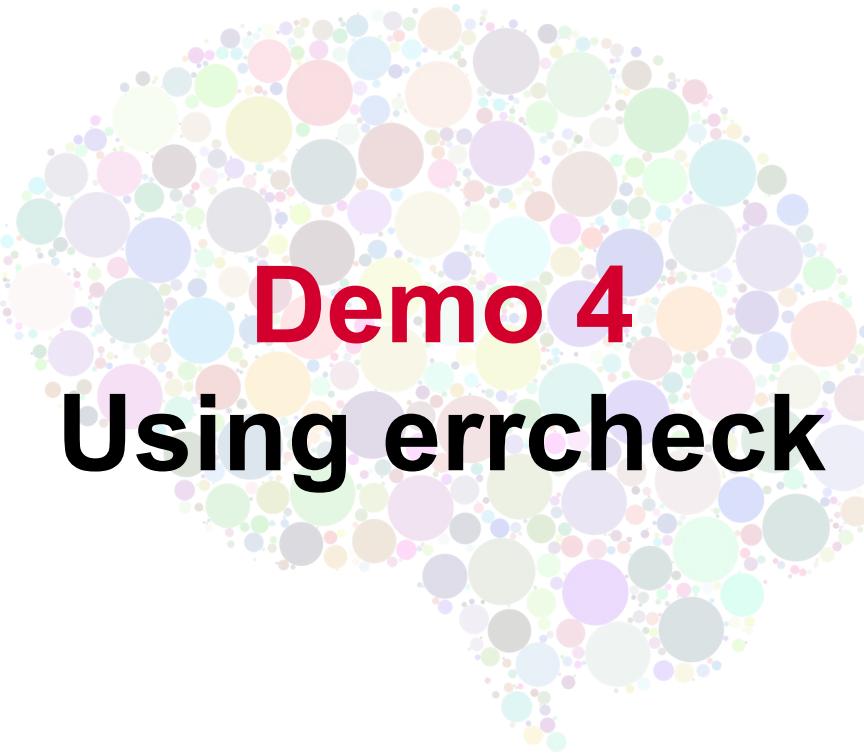
Checking for unchecked errors

github.com/kisielk/errcheck

```
$ go get -u github.com/kisielk/errcheck
```

```
$ errcheck ./...
```

*All packages beneath
current directory*



Demo 4

Using errcheck

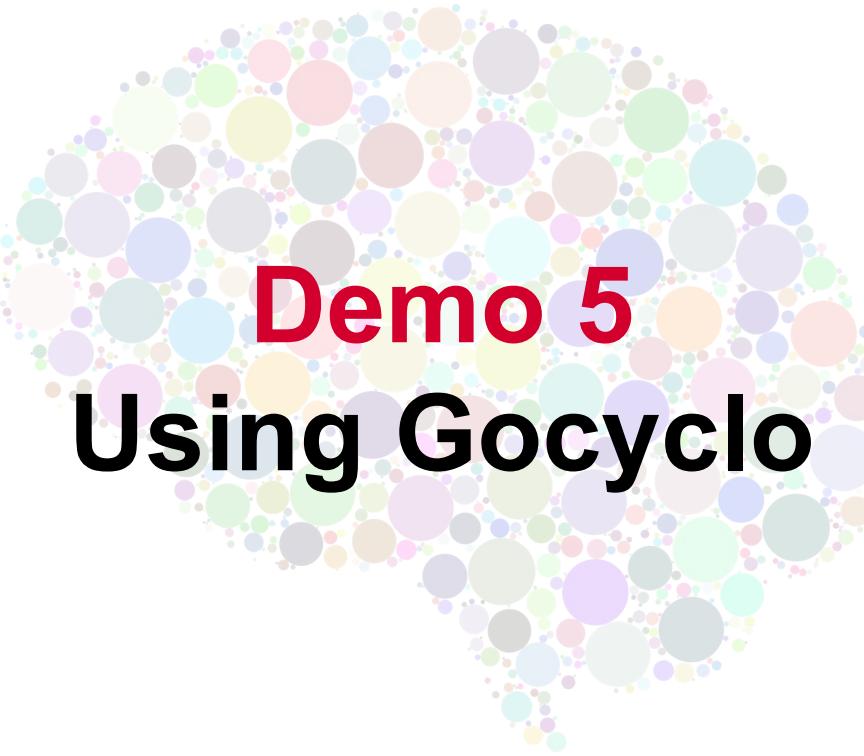
Calculating cyclomatic complexities

github.com/fzipp/gocyclo

```
$ go get -u github.com/fzipp/gocyclo
```

```
$ gocyclo .
```

*All packages beneath
current directory*



Demo 5

Using Gocyclo

Aggregated Linting for Go projects

<https://github.com/golangci/golangci-lint>

Parallel execution of quality tools

Free for Open Source projects



```
$ golangci-lint run
```

IDE integration for golangci-lint



Linting on save

Uses golint by default

Override in settings.json

USER SETTINGS WORKSPACE SETTINGS

Place your settings here to override the Default Settings.

```
1  {
2    "go.lintTool": "golangci-lint",
3    "go.lintFlags": [
4      "--fast"
5    ],
6    "window.zoomLevel": 1,
7    "editor.minimap.enabled": false,
8    "editor.renderWhitespace": "all",
9    "editor.renderControlCharacters": true,
10   "breadcrumbs.enabled": true,
11   "workbench.colorTheme": "Quiet Light"
12 }
```

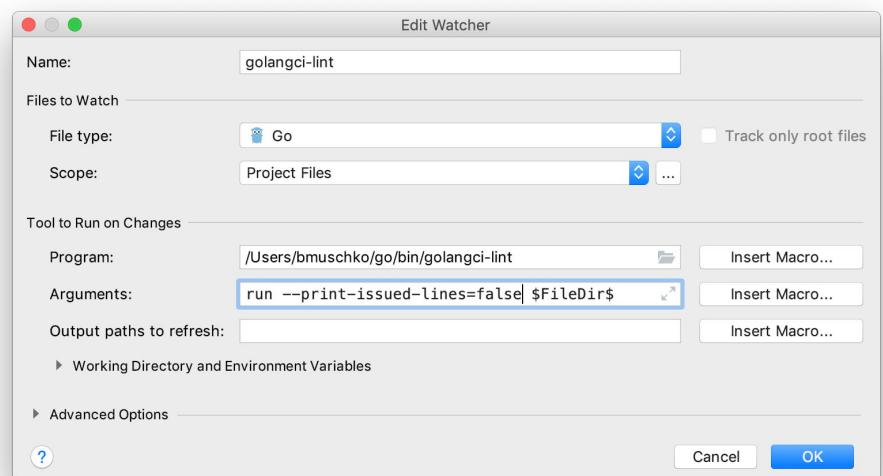
IDE integration for golangci-lint

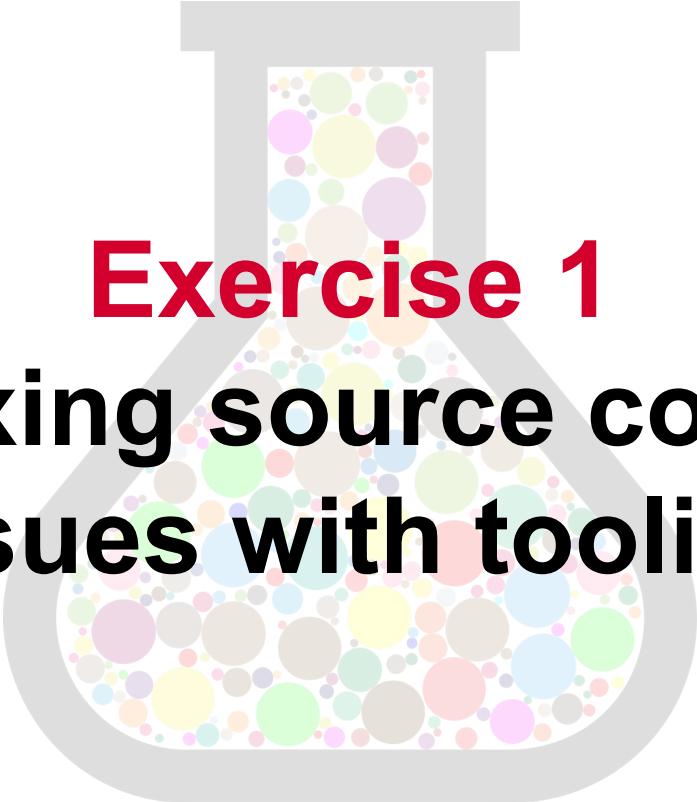


File Watcher

Predefined [File Watcher](#) for
golangci-lint

Available with >= 2019.1.1





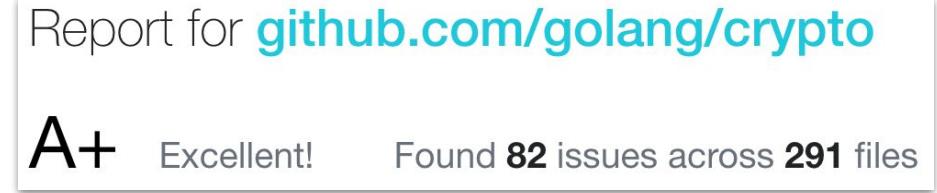
Exercise 1

Fixing source code issues with tooling

Generating a project report

goreportcard.com/

Results
gofmt 99%
go_vet 98%
gocyclo 84%
golint 83%
license 100%
ineffassign 97%
misspell 100%



Full integration
with different tools

Report
summary

go report A+

Embeddable badge



Demo 6

Using Go report card

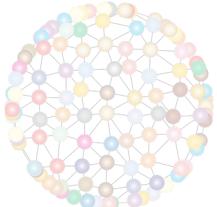
Package management

2



Discussion

What kind of package management do you use?



Options for package managers

A lot of open source choices: glide, godep, ...

Most relevant: [dep](#) and [Go Modules](#)

The new standard (>= Go 1.11): [Go Modules](#)

Using an external package

```
package cmd

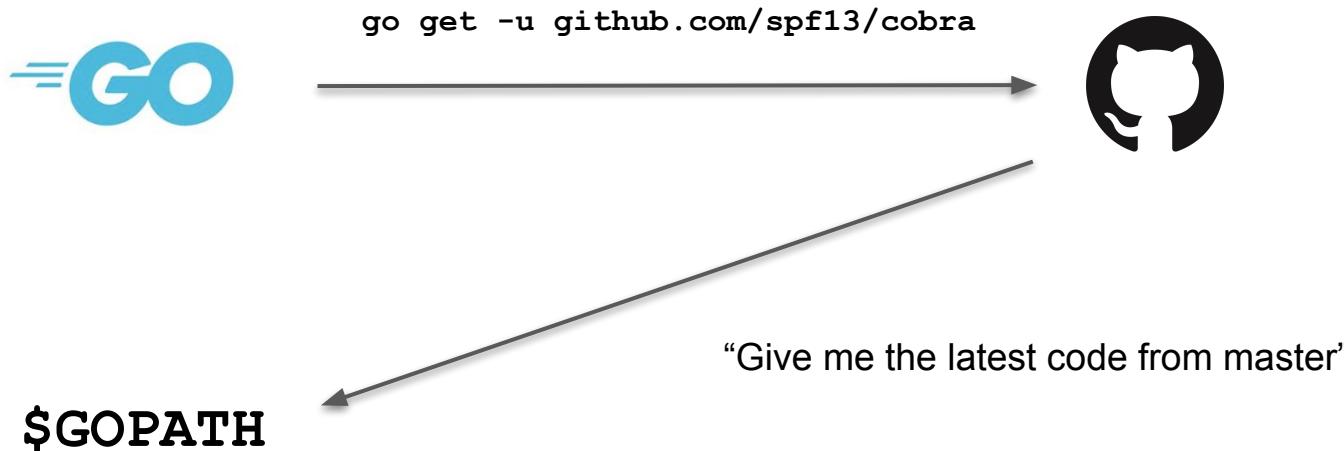
import (
    "fmt"
    "github.com/spf13/cobra"
    "os"
)
```



References package outside of
Go's internal packages

Package management

Typical workflow of go get (< Go 1.11)



Shortcomings of go get (< Go 1.11)

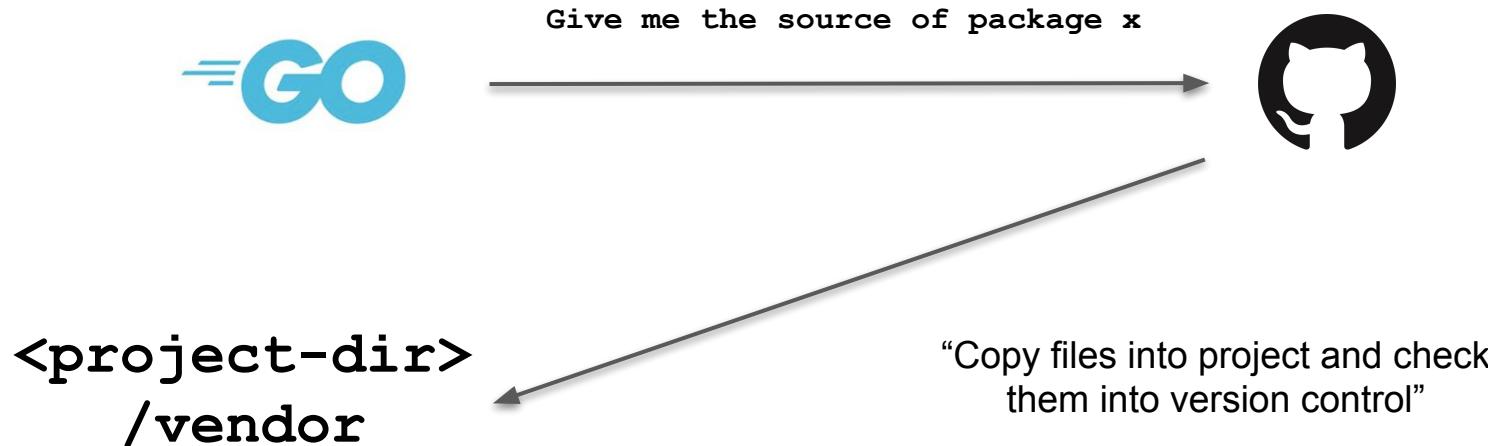
“The code is currently broken on master”

“Will I get the same code in 3 months?”

“What about transitive packages?”

Package management

Vendoring to the rescue



Go Modules - high level usage

Enabled by setting `GO111MODULE=on`

Check in generated `go.mod`, `go.sum`

Direct integration with `go get`

Semantic versioning to identify release

semver.org/

1 . 3 . 12



Major

(Breaking)

Minor

(Feature)

Patch

(Fix)

Package management

Dependency declaration

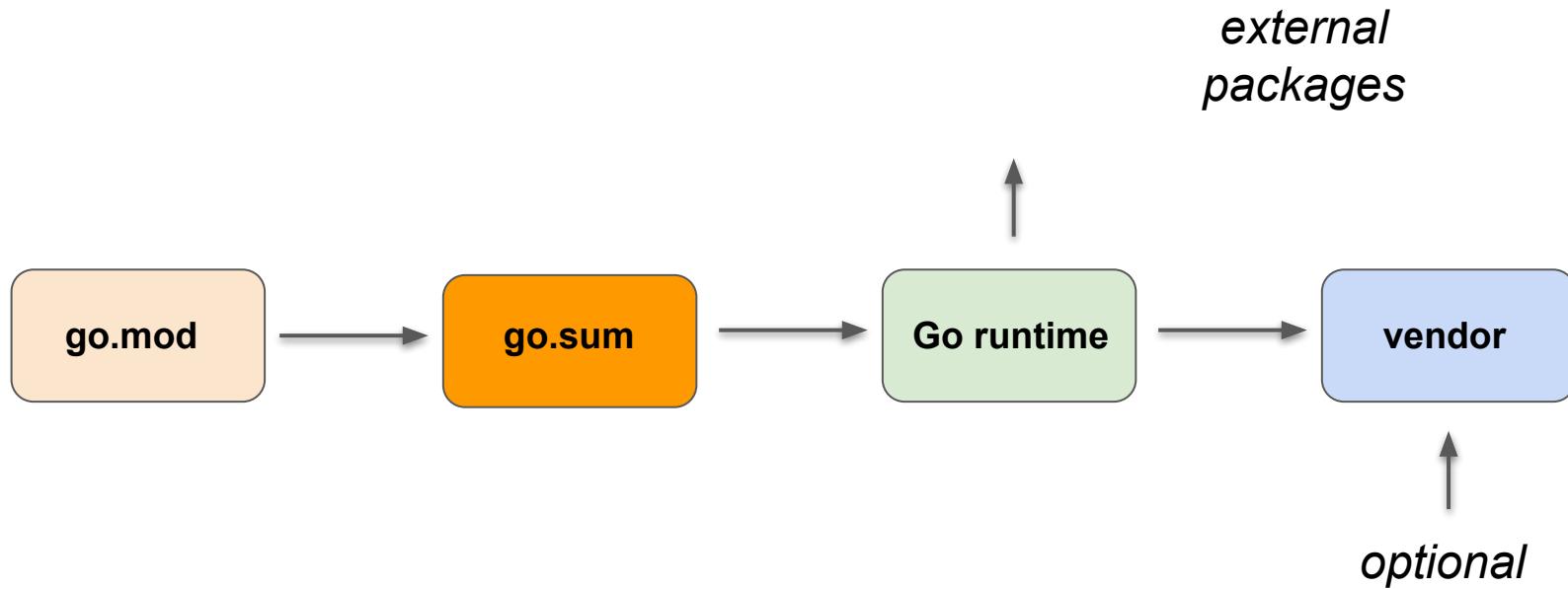
```
module github.com/bmuschko/letsgopher

require (
    github.com/Flaque/filet v0.0.0-20190209224823-fc4d33cf93
    github.com/blang/semver v3.5.1+incompatible
    github.com/ghodss/yaml v1.0.0
    github.com/gosuri/uitable v0.0.0-20160404203958-36ee7e946282
    github.com/inconshreveable/mousetrap v1.0.0 // indirect
    ...
)
go 1.13
```

go.mod

Package management

Building blocks



Workflow: Starting a new project

1. Create new project, set env. variable
2. Run `go mod init`, add pkg with `go get`
3. Check in generated `go.mod`, `go.sum`

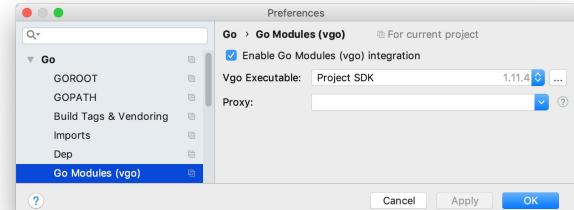
IDE integration for Go modules



Early support, change of settings.json



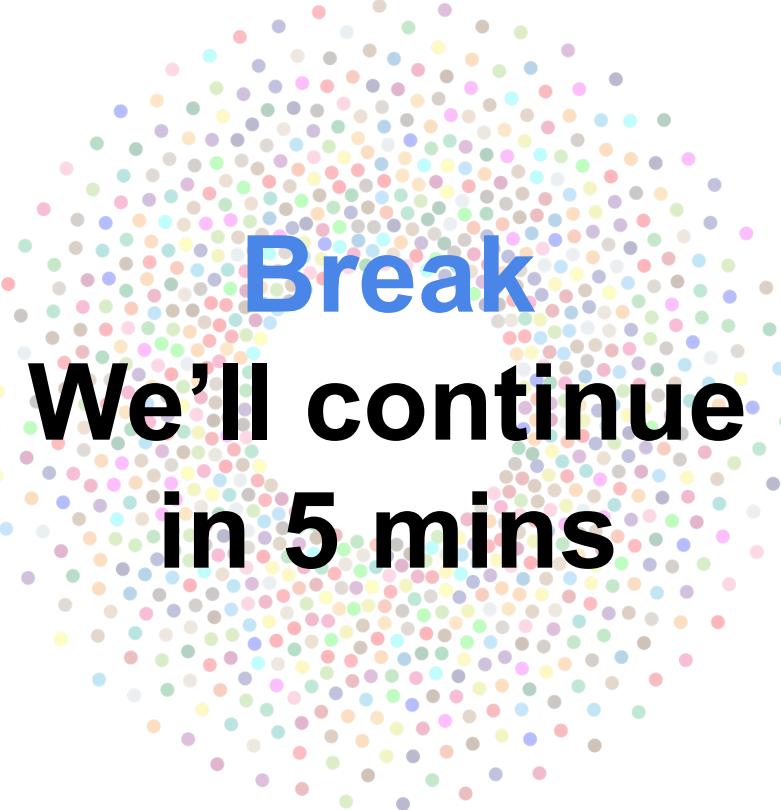
Built-in support can run even run mod commands





Exercise 2

Declaring and resolving dependencies



Break
We'll continue
in 5 mins

Efficient testing

3



Discussion

What kind of tests do you write? Do you use testing frameworks?

Using the standard testing package

Built-in feature of Go library

Easy to understand, no syntactic sugar

No convenient assertion statements

Basic guidelines for organizing tests

Create test file alongside prod. source code

Use convention `*_test.go` for test file name

Only test public API by using different package

Example of test file organization



Implementing a simple test

```
package calc_test

import (
    . "github.com/bmuschko/go-testing-frameworks/calc"
    "testing"
)

func TestAddWithTestingPackage(t *testing.T) {
    result := Add(1, 2)

    if result != 3 {
        t.Errorf("Result was incorrect, got: %d, want: %d.", result, 3)
    }
}
```

Using the testing API

Test function name starts with `Test`

Only parameter for function is `t *testing.T`

Call `t.Error` or `t.Fail` to indicate a failure

Executing tests including coverage

```
$ go test ./... -cover
?    github.com/bmuschko/link-verifier      [no test files]
?    github.com/bmuschko/link-verifier/cmd   [no test files]
ok   github.com/bmuschko/link-verifier/file  0.023s  coverage: 94.7%
of statements
ok   github.com/bmuschko/link-verifier/http   0.536s  coverage: 100.0%
of statements
ok   github.com/bmuschko/link-verifier/stat   0.020s  coverage: 100.0%
of statements
ok   github.com/bmuschko/link-verifier/text   0.035s  coverage: 100.0%
of statements
?    github.com/bmuschko/link-verifier/verify [no test files]
```

IDE integration in VSCode



Ability to run and debug tests

```
run test | debug test
func TestAddWithTestingPackage(t *testing.T) {
    result := Add(1, 2)

    if result != 3 {
        t.Errorf("Result was incorrect, got: %d, want: %d.", result, 3)
    }
}
```

run package tests | run file tests
package · calc_test

- >
- Go: Test All Packages In Workspace
- Go: Test File
- Go: Test Package

IDE integration in GoLand



Ability to run and debug tests



A screenshot of the GoLand IDE interface. On the left, the code editor displays a Go test function:

```
func TestAddWithTestingPackage(t *testing.T) {
    result := Add(1, 2)

    if result != 3 {
        t.Errorf("Result was incorrect, got: %d, want: %d.", result, 3)
    }
}
```

The right side shows a toolbar with several test-related options:

- Run 'go test github.com/b...'
- Debug 'go test github.com/b...'
- Run 'go test github.com/b...' with Coverage
- go test github.com/b...
- gobench github.com/b...



Exercise 3

Implementing a simple test

Capturing code coverage metrics

```
$ go test ./... -coverprofile=coverage.txt -covermode=count
ok    github.com/bmuschko/go-testing-frameworks/calc  3.610s
```

Vendor directory excluded with Go 1.9+

Write metrics to `coverage.txt` for processing

Rendering coverage HTML report

```
$ go tool cover -html=coverage.txt
```



```
github.com/bmuschko/lets-gopher-exercise/utils/file.go (7.3%)
not tracked no coverage low coverage * * * * * * * * * high coverage
package utils
```

```
import (
    "fmt"
    "io"
    "io/ioutil"
    "os"
    "path"
)

func CreateDir(dir string) error {
    if _, err := os.Stat(dir); os.IsNotExist(err) {
        if os.MkdirAll(dir, 0755) != nil {
            return err
        }
    }
    return nil
}

func CopyFile(src, dst string) error {
    var err error
    var srcfd *os.File
    var dstfd *os.File
    var srcinfo os.FileInfo

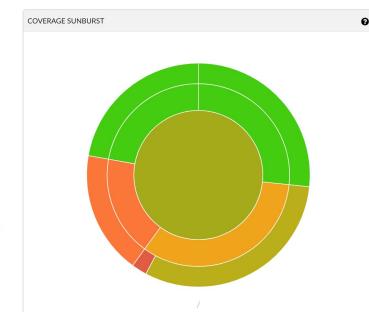
    if srcfd, err = os.Open(src); err != nil {
        return err
    }
```

Honestly not the most visually appealing!

Coverage visualization with Codecov

codecov.io/

Files	30	24	2	4	Coverage
file	30	24	2	4	80.00%
http/http.go	16	12	1	3	75.00%
stat/stat.go	20	20	0	0	100.00%
text/parse.go	24	24	0	0	100.00%
Project Totals (5 files)	90	80	3	7	88.88%



```
$ bash <(curl -s https://codecov.io/bash)
```

Other testing frameworks

Comparison: bmuschko.com/blog/go-testing-frameworks/

Testify: Assertion and mock helper functions

Ginkgo: BDD framework + assertion helpers

Benefits of Testify

Reduction of code duplication

Readable and standardized error messages

Creation of stand-in objects

Testify comparison assertion

godoc.org/github.com/stretchr/testify/assert#Equal

func Equal

```
func Equal(t TestingT, expected, actual interface{}, msgAndArgs ...interface{}) bool
```

Equal asserts that two objects are equal.

```
assert.Equal(t, 123, 123)
```

Pointer variable equality is determined based on the equality of the referenced values (as opposed to the memory addresses). Function equality cannot be determined and will always fail.

Using a Testify assertion

```
package calc_test

import (
    . "github.com/bmuschko/go-testing-frameworks/calc"
    . "github.com/stretchr/testify/assert"
    "testing"
)

func TestAddWithTestify(t *testing.T) {
    result := Add(1, 2)
    Equal(t, 3, result)
}
```



Exercise 4

Implementing and running tests using Testify

Benefits of Ginkgo

Expressive test definition (given/when/then)

Optional assertion library Gomega Ω

Color-coded console output



Implementing a scenario with Ginkgo

```
package calc_test

import (
    . "github.com/bmuschko/go-testing-frameworks/calc"
    . "github.com/onsi/ginkgo"
    . "github.com/onsi/gomega"
)

var _ = Describe("Calculator", func() {
    Describe("Add numbers", func() {
        Context("1 and 2", func() {
            It("should be 3", func() {
                Expect(Add(1, 2)).To(Equal(3))
            })
        })
    })
})
```

Defining a test suite

```
package calc_test

import (
    "testing"
)

func TestCalc(t *testing.T) {
    RegisterFailHandler(Fail)
    RunSpecs(t, "Calculator Suite")
}
```

Color-coded console output

```
== RUN TestCalc
Running Suite: Calculator Suite
=====
Random Seed: 1528760387
Will run 4 of 4 specs

....
Ran 4 of 4 Specs in 0.000 seconds
SUCCESS! -- 4 Passed | 0 Failed | 0 Pending | 0 Skipped
--- PASS: TestCalc (0.00s)
```

Building artifacts

4

Rendering docs on local machine

Documentation server on localhost

Renders anything in all packages in GOPATH

Option to render docs on console

```
$ godoc -http=:6060
```

Rendering docs from CLI

Docs for package, const, func, type, var, method

Primary and only use from CLI

```
$ go doc stat.SumFailures
```

Building artifacts

The godoc.org website

```
// Options represent command line options exposed by this program.  
type Options struct {  
    RootDirs      []string  
    IncludePatterns []string  
    Fail          bool  
    Timeout       int  
}
```

The diagram illustrates the process of generating public documentation for a Go package. On the left, a snippet of Go code defines a type `Options` with fields for `RootDirs`, `IncludePatterns`, `Fail`, and `Timeout`. An arrow points from this code to the right, where a screenshot of the godoc.org website shows the generated documentation. The documentation page includes a header with `GoDoc`, `Home`, and `About` links, a search bar, and navigation links for `link-verifier`, `Index`, and `Files`. The main content displays the `cmd` package, its import path, and an `Index` section containing the `Options` type definition. The code for `Options` is identical to the original snippet, demonstrating that the generated documentation is a direct reflection of the source code.

package cmd
import "github.com/bmuschko/link-verifier/cmd"

Index
type Options
 ◦ func ParseOptions() Options

Package Files
cmd.go

type Options

```
type Options struct {  
    RootDirs      []string  
    IncludePatterns []string  
    Fail          bool  
    Timeout       int  
}  
  
Options represent command line options exposed by this program.
```

*Publicly-searchable
documentation for go packages*

Assembling cross-platform binaries

Executables for platforms & architectures

Optimized for performance & includes deps

go build + GOOS and GOARCH env vars

Options for creating binaries

Write a [script](#) to automate CLI invocations

[Gox](#): Parallelize builds for multiple platforms

[GoReleaser](#): Binary creation and releasing

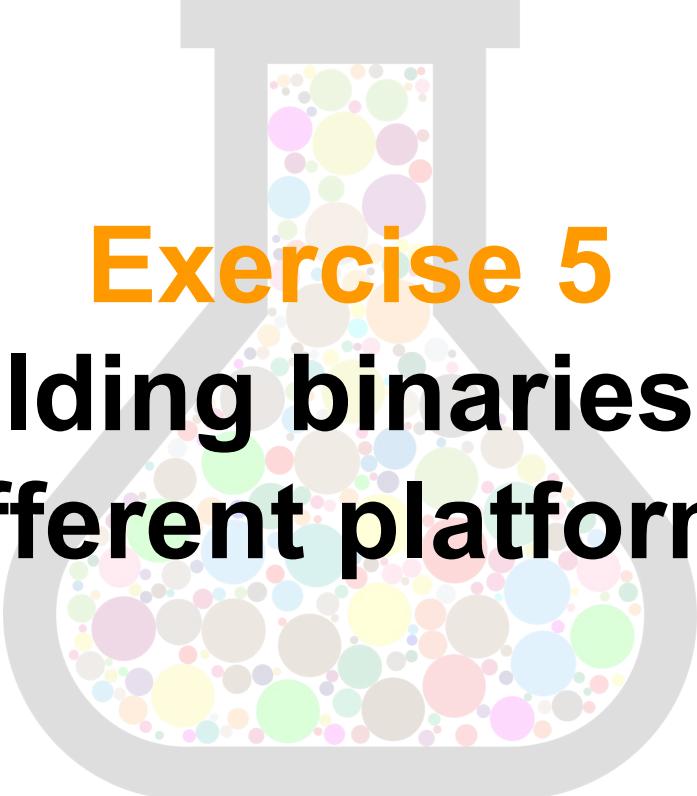
Simplifying cross-compilation with Gox

github.com/mitchellh/gox

Supports all possible platforms

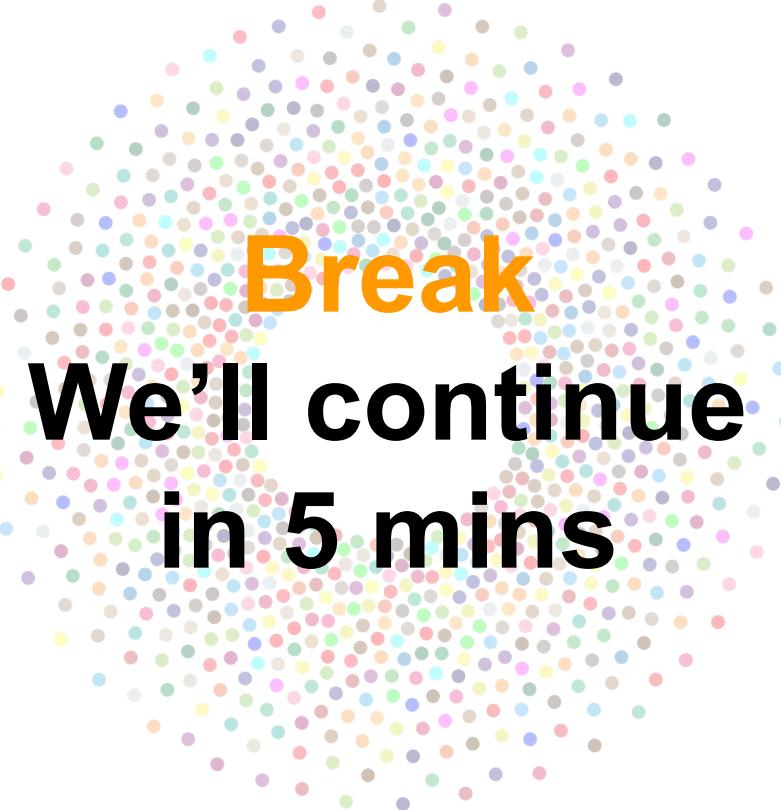
Parallelized execution based on CPUs/cores

```
$ gox -osarch="linux/amd64"
```



Exercise 5

Building binaries for different platforms



Break

**We'll continue
in 5 mins**

Publishing artifacts

5

Making executables consumable

Versioned binaries for end users

Appropriate hosting platform

An option for public projects: GitHub releases

Publishing artifacts

GitHub releases

Tag version

Release title

Binaries

Release notes



Releases Tags Draft a new release

Latest release v0.4 bccf2a9

0.4

bmuschko released this on May 14 - 16 commits to master since this release

Assets 14

Asset	Size
link-verifier-0.4-arm.tar.gz	1.46 MB
link-verifier-0.4-freebsd32.tar.gz	1.48 MB
link-verifier-0.4-freebsd64.tar.gz	1.56 MB
link-verifier-0.4-linux32.tar.gz	1.48 MB
link-verifier-0.4-linux64.tar.gz	1.56 MB
link-verifier-0.4-netbsd32.tar.gz	1.48 MB
link-verifier-0.4-netbsd64.tar.gz	1.56 MB
link-verifier-0.4-openbsd32.tar.gz	1.48 MB
link-verifier-0.4-openbsd64.tar.gz	1.56 MB
link-verifier-0.4-osx.tar.gz	1.65 MB
link-verifier-0.4-win32.zip	1.43 MB
link-verifier-0.4-win64.zip	1.52 MB
Source code (zip)	
Source code (tar.gz)	

Different tooling options

[ghr](#): Creates GitHub release/uploads in parallel

[github-release](#): Create/delete releases on Github

[GoReleaser](#): Builds/release binaries for platforms

Using GoReleaser to publish to GitHub

Definition in `.goreleaser.yml`

Combination of OS and architecture

Automatic generation of changelog





Exercise 6

Publishing binaries to GitHub

Publishing to binary repository

Hosting artifacts within enterprise boundaries

[Artifactory](#) for ensuring security & confidentiality

GoReleaser integrates with Artifactory

Using GoReleaser to publish to Artifactory

```
release:  
  disable: true  
artifactories:  
- name: production  
  target: http://localhost:8081/artifactory/←  
    example-repo-local/{{ .ProjectName }}/←  
    {{ .Version }}/  
  username: admin
```

Publishing artifacts

Providing credentials as env. variables

*ARTIFACTORY_
PRODUCTION_
USERNAME*



*ARTIFACTORY_
PRODUCTION_
SECRET*

The screenshot shows the JFrog Artifactory interface. On the left is a tree view of repositories: 'example-repo-local' is expanded, showing 'link-verifier/0.6.1'. This folder contains multiple tar.gz files for various platforms. On the right is a detailed view of a specific artifact: 'link-verifier-0.6.1-darwin-386.tar.gz'. The details pane shows the following information:

General	Properties
Name: link-verifier-0.6.1-darwin-386.tar.gz	Repository Path: example-repo-local/link-verifier/0.6.1/link-verifier-0.6.1-darwin-386.tar.gz
Module ID: N/A	
Deployed By: admin	
Size: 1.63 MB	
Created: 28-12-18 15:02:16 -06:00	
Last Modified: 28-12-18 15:02:16 -06:00	
Downloads: 0	
Remote Downloads: 0	
Checksums	
SHA-256:	f9c0511a19b93751b3e6b4d954d9487385b20f9f...
SHA-1:	b31e2b13c9ffd4f56afa5db348bd560d7127bd2d ...



Demo 7

Publishing artifacts to Artifactory

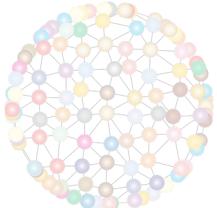
CI/CD for Go projects

6



Discussion

What's your approach to project automation?



On the importance of project automation

Avoid manual steps, reproducibility

Continuous integration of code commits

Ensuring a releasable state of binaries

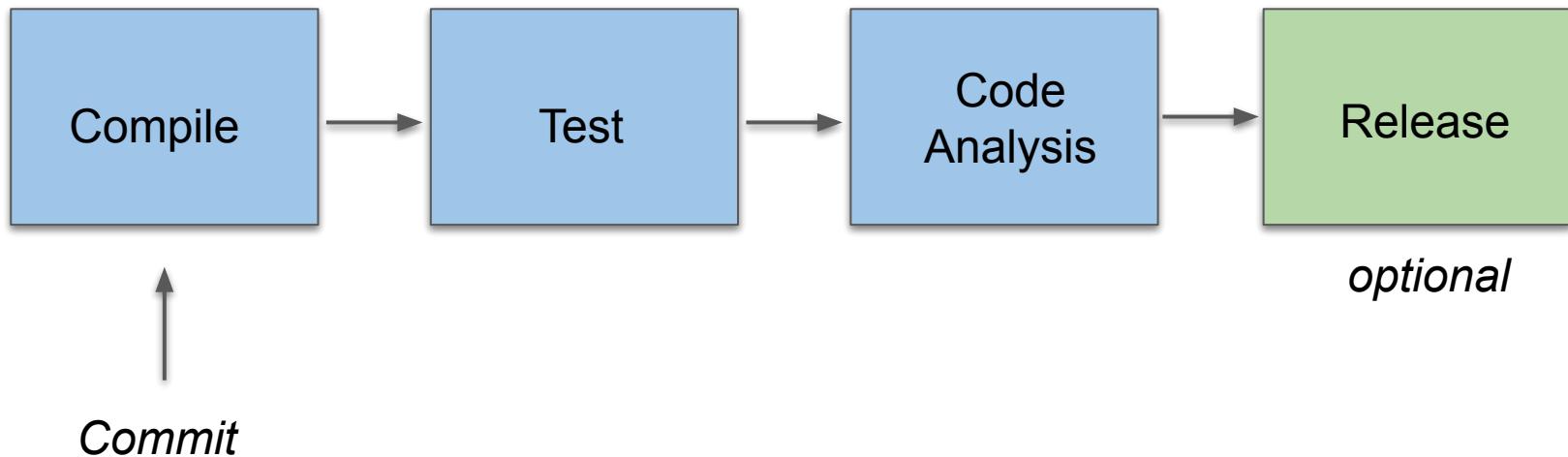
Different tooling options

Travis CI: Hosted free and paid on-prem solution

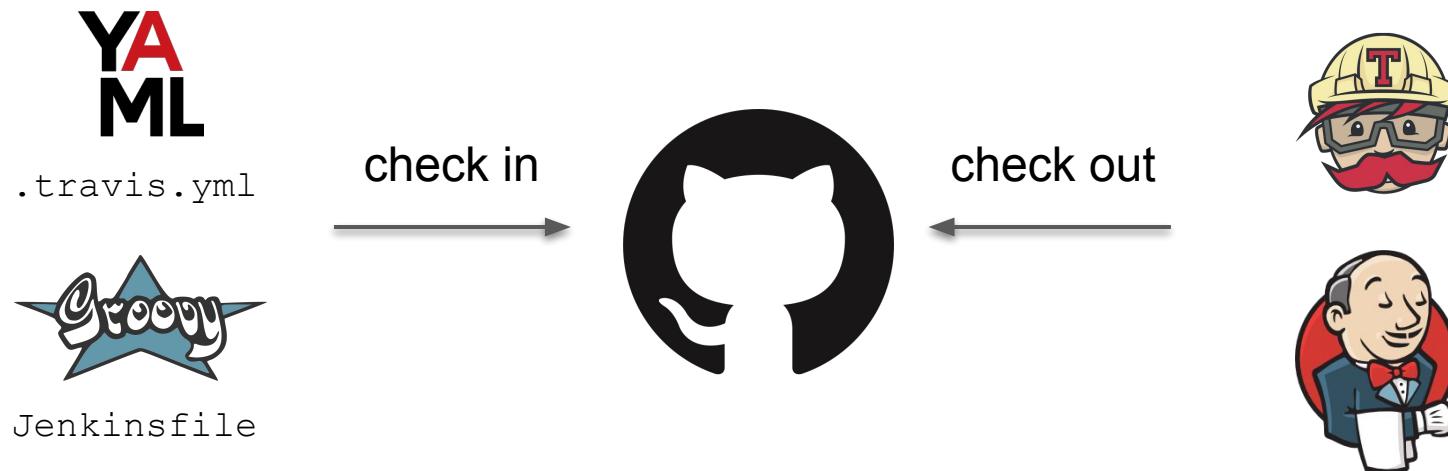
Jenkins: Free and commercial solution

...and many others like Drone, Circle CI

The delivery pipeline we want to model



Modeling a build pipeline as code



Language support and Go Module tooling

```
language: go

go:
  - 1.13.x

install: true

cache:
  directories:
    - $GOPATH/pkg/mod
```



Compiling packages and dependencies

script:

- go build



Running tests with code coverage

script:

- go test ./... -coverprofile=coverage.txt



after_success:

- bash <(curl -s https://codecov.io/bash)

Executing code analysis

before_script:

- curl -sfL https://install.goreleaser.com
/github.com/golangci/golangci-lint.sh
| bash -s -- -b \$GOPATH/bin v1.21.0



script:

- golangci-lint run

Release binary for tagged commit

```
deploy:  
  - provider: script  
    skip_cleanup: true  
    script: curl -sL https://git.io/goreleaser | bash  
  on:  
    tags: true  
    condition: $TRAVIS_OS_NAME = linux
```





Using build stages

Group jobs and run them sequentially

Grouped jobs can be run in parallel

Defined in `.travis.yml`

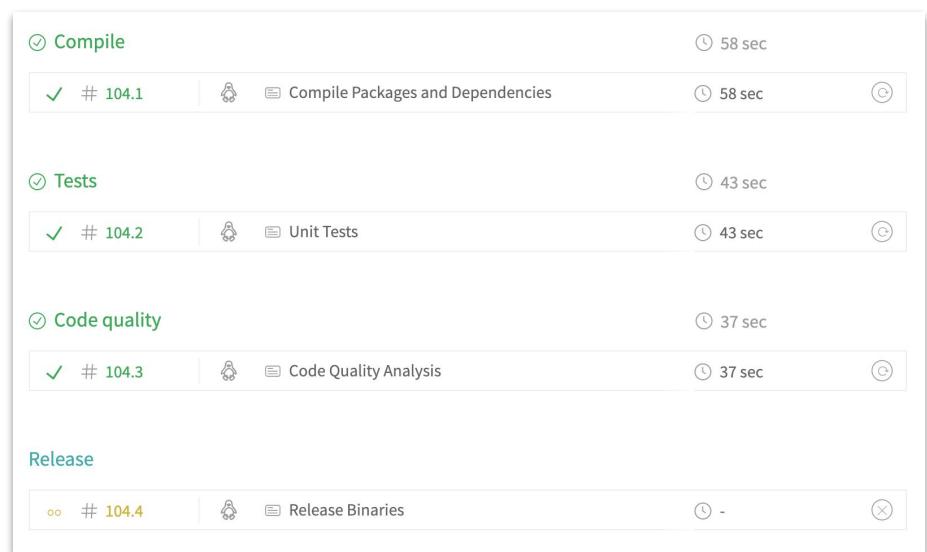
CI/CD for Go projects

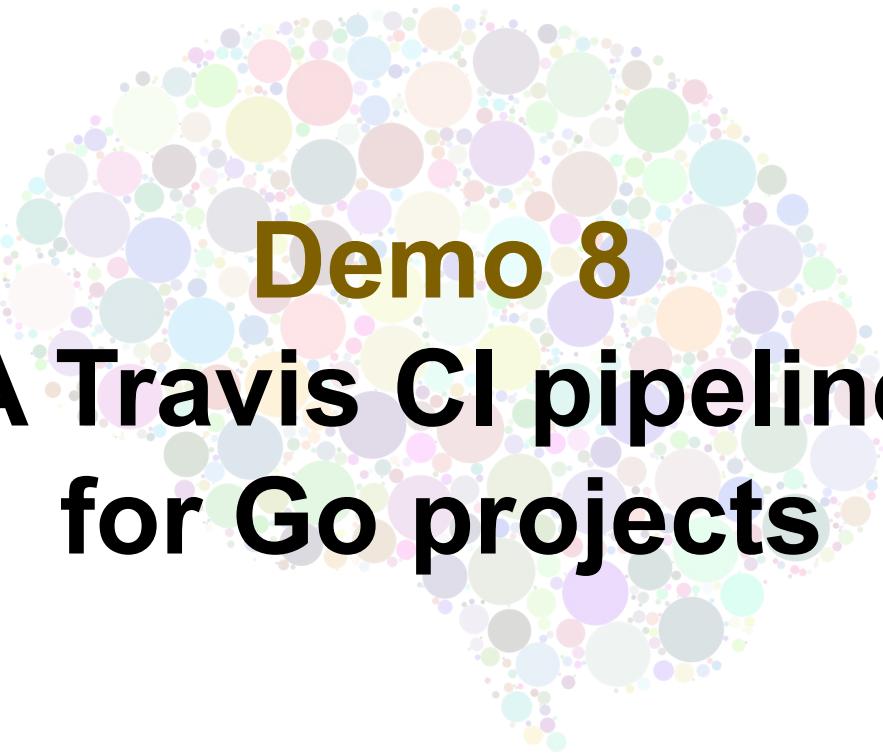


Visualization of build pipeline

.travis.yml

```
jobs:  
  include:  
    - stage: ...  
    - stage: ...  
    - stage: ...
```





Demo 8

A Travis CI pipeline for Go projects



Automatic Go runtime installation

Taken care of by [Jenkins Go plugin](#)

Global Tool Configuration
Configure tools, their locations and automatic installers.

Go

Go installations

Add Go

Go

Name: go-1.11

Install automatically

Install from golang.org

Version: Go 1.11.4

Delete Installer

This screenshot shows the Jenkins Global Tool Configuration interface for the Go plugin. It displays a configuration page for managing Go tool installations. A specific installation named 'go-1.11' is shown, which has been installed automatically from golang.org. The version listed is Go 1.11.4. There are buttons for adding new installations and deleting existing ones.

Go tooling and environment

```
pipeline {
    agent any

    tools {
        go 'go-1.13'
    }

    environment {
        GO111MODULE = 'on'
    }
}
```



Compilation and resolving dependencies

```
pipeline {  
    ...  
  
    stages {  
        stage('Compile and Resolve Dependencies') {  
            steps {  
                sh 'go build'  
            }  
        }  
    }  
}
```



Running tests with code coverage

```
stage('Test') {
    environment {
        CODECOV_TOKEN = credentials('codecov_token')
    }
    steps {
        sh 'go test ./... -coverprofile=coverage.txt'
        sh 'curl -s https://codecov.io/bash | bash -s -'
    }
}
```



Scope	Global (Jenkins, nodes, items, all child items, etc)
Secret
ID	codecov_token
Description	Codecov Token

Executing code analysis

```
stage('Code Analysis') {
    steps {
        sh 'curl -sfL https://install.goreleaser.com/←
            github.com/golangci/golangci-lint.sh | bash -s←
            -- -b $GOPATH/bin v1.21.0'
        sh 'golangci-lint run'
    }
}
```



Release binary for tagged commit

```
stage('Release') {
    when {
        buildingTag()
    }
    environment {
        GITHUB_TOKEN = credentials('github_token')
    }
    steps {
        sh 'curl -sL https://git.io/goreleaser | bash'
    }
}
```



Scope	Global (Jenkins, nodes, items, all child items, etc)
Secret
ID	github_token
Description	GitHub Token

CI/CD for Go projects



Visualization of build pipeline



Standard

Blue Ocean



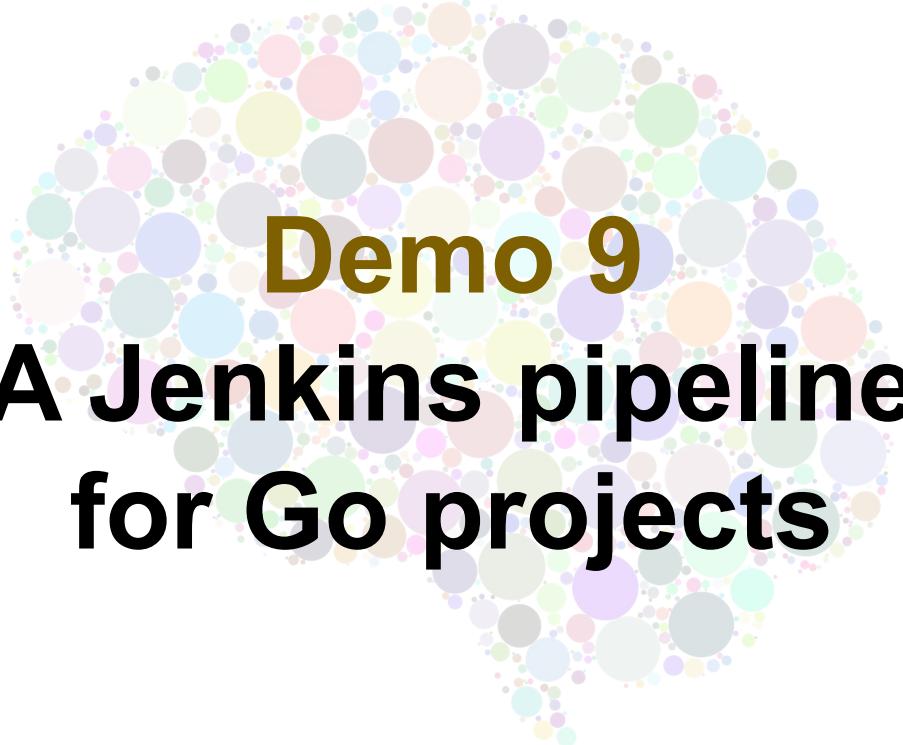
Jenkinsfile IDE support



[Jenkins Pipeline Linter Connector](#)



[Jenkinsfile language support](#)



Demo 9

A Jenkins pipeline for Go projects

Sample pipeline definitions

Travis CI: using [Go modules](#), using [dep](#)

Jenkins: using [Go modules](#), using [dep](#)

